Explicit modelling of physical measures: from Event-B to Java

John Paul Gibson, Dominique Méry

Explicit Modelling of Physical Measures: From Event-B to Java

J Paul Gibson
SAMOVAR UMR 5157
Télécom Sud Paris, Evry, France
paul.gibson@telecom-sudparis.eu

Dominique Méry
LORIA UMR 7503
Telecom Nancy, Université de Lorraine
Vandœuvre-lès-Nancy, France
dominique.mery@loria.fr

The increasing development of cyber-physical systems (CPSs) requires modellers to represent and reason about physical values. This paper addresses two major, inter-related, aspects that arise when modelling physical measures. Firstly, there is often a *heterogeneity of representation*; for example: speed can be represented in many different units (mph, kph, mps, etc...). Secondly, there is *incoherence in composition*; for example: adding a speed to a temperature would provide a meaningless result in the physical world, even though such a purely mathematical operation is meaningful *in the abstract*. These aspects are problematic when implicit semantics — concerned with measurements — in CPSs are not explicit (enough) in the requirements, design and implementation models. We present an engineering approach for explicitly modelling measurements during all phases of formal system development. We illustrate this by moving from Event-B models to Java implementations, via object oriented design.

1 Introduction

1.1 Problems when using implicit measurements and dimensions

Historically, humans have made many critical mistakes when using units of measurement. For example, Chrostopher Columbus miscalculated the circumference of the earth when he used Roman miles instead of nautical miles [24]. Since software has been written, there is added risk of bugs arising out of the lack of explicit measurement types (and dimensions) in programs. Perhaps the best known bug of this type is the Mars climate orbiter error [25], where separate software modules were using different units of measurement (imperial and metric); yet when they communicated data they were not aware of the different representations and wrongly assumed that the data being shared was represented in the same unit/scale. This issue would have been avoided if the units of measurement had been explicitly (and formally) specified in the module interfaces.

1.2 Proposed requirements for explicit modelling of measures

Formal methods are often used in the development of critical software systems [6]; but they often permit the construction of models which are mathematically meaningful, in the abstract, but which have no coherent interpretation in the real (concrete) world. We propose a modelling technique which helps software engineers to avoid building models that are incoherent, by explicitly specifying measurement dimensions and scales (units).

We are motivated by the international standard for physical measurements which identifies seven base units/dimensions and numerous derived units and constants [23, 27]. We require that the measurement model must: **(1)** Specify seven base units (explicit) in a standard library: mass, length, time, temperature,

R. Laleau, D. Méry, S. Nakajima, E. Troubitsyna (Eds): Joint Workshop on Handling IMPlicit and EXplicit knowledge in formal system development and Formal & Model-Driven Techniques for Developing Trustworthy Systems.
EPTCS 271, 2018, pp. 64–79, doi:10.4204/EPTCS.271.5

current, light and matter. **(2)** Support the derivation of (standard) units that can be added explicitly to the library (as required) or used implicitly. **(3)** Support definition of non-standard units that can be defined in terms of standard units. **(4)** Facilitate specification of scalar universal constants (like PI). **(5)** Facilitate specification of physical constants of certain dimension (like the speed of light in a vacuum). **(6)** Be amenable to static/automated analysis for detection of incoherent mixing of units (like adding a speed and a temperature). **(7)** Support implicit scaling - being able to add/subtract/compare measurements in the same dimension but different scales, without the need to explicitly scale them.

We choose to use Event-B [1] contexts as our modelling language because we wish to re-use our measurement models to play the role of a domain ontology [14] in the development of safety-critical CPSs using refinement [3]. We also wish to have a powerful reasoning tool (in this case Rodin [2]) for automating validation and verification. Our modelling approach could be easily followed using other formal methods (that provide similar language and tool support).

2 Related work

The thesis by Kennedy [19] is one of the earliest works to propose a solution to the problem of representing dimensions in programming languages. The thesis establishes and proves a formal relationship between conventional type systems and the dimension type systems (using a functonal programming language as its foundations). However, the work does not not cover the situation of multiple systems of units within the same dimension (formal engineering requirements 3 and 7, in section 1.2).

In '*Object-Oriented Units of Measurement*' [4], the authors state: *"Physical units and dimensions are a commonly used computational construct in science and engineering, but there is relatively little support for them in programming languages."* They demonstrate how an OO language that offers rich semantics for polymorphism and dynamic binding can use explicit typing to model a hierachy of measurements for physical values. Their approach also does not meet requirements 3 and 7.

In '*Arithmetic with Measurements on Dynamically-typed Object-oriented Languages*' [28], the authors implement - in Smalltalk - a measurement system that is general enough to include the SI units for physical measurements, as well as non-physical measures such as units of currency (see section 5). The choice of Smalltalk is interesting because the language's dynamic type system gives the programmer more support for defining new types, but restricts the strength of static analysis that can be done. Their approach does not support our requirements 2,4, and 5.

In '*Osprey: A Practical Type System for Validating Dimensional Unit Correctness of C Programs*"[17], the authors present a sound type system to automatically check for all potential errors involving units of measurements. The system is based on annotating the C code with unit/dimension typing constraints that are then checked using a constraint solver. However, their analysis is limited to comparing units and does not support comparing dimensions. Their approach only partially meets all our requirements, and introduces complexity of annotations into the programming task.

In "*A model-driven approach to automatic conversion of physical units*" [8], the authors propose checking physical units at the level of the modelling language, removing the need for such a support in the underlying implementation language. They choose to express all physical values in the standard canonical form, as products of powers of the SI base units, and present an elaborate algorithm for translating to and from such a form for any arbitrary unit. However, since in CellML (the modelling language upon which their approach is built) all constants and variables are explicitly annotated with their units, they have not needed to make any type inferences. Thus, they have not considered meeting our 2nd requirement.

3 Formal Specification in Event-B

In this section we present three different Event-B contexts which are used to meet our measurement modelling requirements, as specified in subsection1.2. We chose to structure the model by separating the representation of real-world values (in context `C_Values`) from the real-world dimensions (in context `C_Dimensions`). The context `C_Measurements` then composes them to provide a standard model that can then be extended with derived measurements and values.

3.1 Values as floats

We define a context `C_Values` to provide a concrete mechanism for constructing and representing values as floating point decimal numbers [13]. A partial specification of the model is given in Figure 1.The constant `PI` is used as a concrete example of how we represent values using floating point numbers. The first integer, commonly know as the *significand*, is to be interpreted as a float with the floating point occurring after the first decimal digit. The second integer is to be interpreted as the power of 10, commonly known as the *base*, which is to be multiplied to the *significand* in order to give the real value of the floating point number ($significand \times 10^{base}$). The arithmetic operators are defined algebraically (including axioms - not shown - for operator properties, such as commutativity, that are needed for later proofs). We note that this context makes no reference to physical units or dimensions.

```
CONTEXT
  C_Values
SETS
  Values
CONSTANTS
  MakeFloat
  PI
  addValues
  subtractValue
  multiplyValues
  divideValue
  gt      //   greater than
  lt      //   less than
  eq      //   equality
AXIOMS
  float constructor   :   MakeFloat ∈ ℤ × ℤ ⟶ Values       //   significand X base
  PI type   :   PI ∈ Values
  PI value   :   PI = MakeFloat ( 314159265358979 ↦ 0)      //   3.14159265358979 X 10 ^ 0
  addition   :   addValues  ∈ Values × Values ⟶ Values
  subtraction   :   subtractValue  ∈ Values × Values ⟶ Values
  multiplication   :   multiplyValues  ∈ Values × Values ⟶ Values
  division   :   divideValue  ∈ Values × Values ⇸ Values       //   not defined for division by 0
  comparison   :   partition((Values × Values), gt, lt, eq)
 ... // AXIOMS for algebraic properties (like commutativity)
 END
```

Figure 1: The floating number (partial) abstract model

3.2 Dimensions

The partial specification of the context `C_Dimensions` is given in Figure 2.The seven SI dimension units [23, 27] - `Mass`, `Length`, `Time`, `Temperature`, `Light`, `Current` and `Matter` are explicitly modelled, and we can derive new dimensions through multiplication, division and reciprocation. So, for example a derived dimension for `Acceleration` would be defined as: Length divided by Time squared.

So the associated powers would be 1 for `Length` and -2 for `Time`, with all other base dimensions having 0 powers giving: $\{Mass \mapsto 0, Length \mapsto 1, Time \mapsto 2, Temperature \mapsto 0, Light \mapsto 0, Current \mapsto 0, Matter \mapsto 0\}$. We note that this context makes no references to physical values.

```
CONTEXT
  C_Dimensions
SETS
  Dimensions
CONSTANTS
AXIOMS
  7Dims    :   partition (Dimensions, {Mass}, {Length}, {Time}, {Temperature}, {Light}, {Current}, {Matter})
  MassDef    :   MassDimension = { Mass ↦ 1, Length ↦ 0, Time ↦ 0, Temperature ↦ 0, Light ↦ 0, Current ↦ 0, Matter ↦ 0 }
  LengthDef    :   LengthDimension = { Mass ↦ 0, Length ↦ 1, Time ↦ 0, Temperature ↦ 0, Light ↦ 0, Current ↦ 0, Matter ↦ 0 }
  TimeDef    :   TimeDimension = { Mass ↦ 0, Length ↦ 0, Time ↦ 1, Temperature ↦ 0, Light ↦ 0, Current ↦ 0, Matter ↦ 0 }
  TempDef    :   TemperatureDimension = { Mass ↦ 0, Length ↦ 0, Time ↦ 0, Temperature ↦ 1, Light ↦ 0, Current ↦ 0, Matter ↦ 0
  LightDef    :   LightDimension = { Mass ↦ 0, Length ↦ 0, Time ↦ 0, Temperature ↦ 0, Light ↦ 1, Current ↦ 0, Matter ↦ 0 }
  CurrentDef    :   CurrentDimension = { Mass ↦ 0, Length ↦ 0, Time ↦ 0, Temperature ↦ 0, Light ↦ 0, Current ↦ 1, Matter ↦ 0 }
  MatterDef    :   MatterDimension = { Mass ↦ 0, Length ↦ 0, Time ↦ 0, Temperature ↦ 0, Light ↦ 0, Current ↦ 0, Matter ↦ 1 }
  MultDim    :   MultiplyDimensions ∈ Dimension  × Dimension → Dimension
  DivDim    :   DivideDimensions ∈ Dimension  × Dimension → Dimension
  RecipDim    :   ReciprocalDimension ∈ Dimension → Dimension

  ... // AXIOMS for algebraic properties of Dimensions
END
```

Figure 2: The dimension model - `CONTEXT C_Dimensions AXIOMS`

3.3 Measurements

We define a context `C_Measurements` which `EXTENDS C_Values` and `C_Dimensions` and combines them to define the standard SI units for the 7 base dimensions: `Kilogram, Metre, Second, Kelvin, Ampere, Candela`, and `Mol` (see Figure 3). The key modelling decision was to model a measurement as a triple: the dimension, the value, and the bijective normalization function for mapping the value to the standard canonical unit. Thus, the actual unit of measurement is implicit in the function definition. For example, a `Mass` in `Kilograms` is constructed from a `MassDimension`, its `Value` and the `id` function (as it is already in the standard unit canonical form). Thus, `Kilogram(v)` represents a mass of v kg.

```
Kilogram is SI unit for Mass        :   ∀ v· v ∈ Values ⇒ Kilogram(v) = MassDimension ↦ v  ↦  id
Metre is SI unit for Length         :   ∀ v· v ∈ Values ⇒ Metre(v) = LengthDimension ↦ v  ↦  id
Second is SI unit for Time          :   ∀ v· v ∈ Values ⇒ Second(v) = TimeDimension ↦ v  ↦  id
Kelvin is SI unit for Temperature   :   ∀ v· v ∈ Values ⇒ Kelvin(v) = TemperatureDimension ↦ v  ↦  id
Ampere is SI unit for Current       :   ∀ v· v ∈ Values ⇒ Ampere(v) = CurrentDimension ↦ v  ↦  id
Candela is SI unit for Light        :   ∀ v· v ∈ Values ⇒ Candela(v) = LightDimension ↦ v  ↦  id
Mol is SI unit for Matter           :   ∀ v· v ∈ Values ⇒ Mol(v) = MatterDimension ↦ v  ↦  id
```

Figure 3: The 7 SI base unit measurements in `CONTEXT C_Measurements`

We note that we make no assumption that units must be defined by a simple linear function through the origin (as is the case for most physical scales). The measurement model also permits more general affine functions (as for temperature conversion, for example). The advantage of this explicit approach is that we can now formally specify the addition (and subtraction) of `Measurements` as partial functions (which are defined only when the `Measurements` have the same `Dimension`). For simplicity our paper presents - in Figure 4 - only the function for adding measurements. Subtraction is the same problem, whilst multiplication and division require no consistency checking. Similarly, we model partial functions

for comparing measurements, where the functions are defined (using the comparison between the values converted to standard units) only when the measurements are in the same dimensions.

```
Measurement Type   :   Measurements = Dimension × Values × (Values ⤖ Values)
getDimensionType   :   getDimension ∈ Measurements ⟶ Dimension
getDimensionDef    :   ∀d,v,f·   (d ↦ v ↦f ) ∈ Measurements ⇒
                       getDimension( (d ↦ v ↦f ) )= d
getValueType   :   getValue ∈ Measurements ⟶ Values
getFunType   :   getFunction  ∈  Measurements ⟶ (Values ⤖ Values)
getFunDef   :   ∀d,v,f·   (d ↦ v ↦f ) ∈ Measurements ⇒
                getFunction( (d ↦ v ↦f ) )= f
getValueDef   :   ∀d,v,f·   (d ↦ v ↦f ) ∈ Measurements ⇒
                  getValue( (d ↦ v ↦f ) )= v

AdditionMeasurementsType   :   AddMeasurements ∈ Measurements × Measurements ⇸ Measurements      //   defined only when the 2 Measurements
                                                                                                 //    share the same Dimension
                               ∀m1, m2 · m1 ∈ Measurements ∧ m2 ∈ Measurements ∧
                                         getDimension(m1) =  getDimension(m2)  ⇒
                                           AddMeasurements(m1 ↦ m2) =      (  getDimension(m1) ↦
AdditionMeasurementsDef   :                                                   addValues(getValue(m1) ↦
                                                                                    (getFunction(m1)~ (getFunction(m2)(getValue(m2))))) ↦
                                                                              getFunction(m1)
                                                                           )
```

Figure 4: The addition of `Measurements`

The key aspect of the mathematical operators is that the resulting `Measurement` is given in the units of the first `Measurement` parameter. In order to do this, the second parameter `Value` has to be converted to the same units. This is done by converting to the standard base unit of the `Dimension` common to the two parameters, and then converting to the unit of the first parameter. For example the addition of 100 grammes and 2 pounds, would result in $(100 + 907, 18474)$ grammes

4 Illustrating measurement modelling problems

In this section we review a small number of case studies which have included modelling of real-world measures. We compare, with our explicit approach, the way in which the studies (to various degrees) use implicit modelling of units and dimensions.

4.1 Changing the unit of measurement when formalizing a requirements specification

In '*Modeling a Landing Gear System in Event-B*' [20], a refinement is used to introduce timing aspects. The authors note that with respect to time values: '*Since, the type Real is not provided in EventB, we multiply all the data per 10*'. This impliclty suggests that the basic unit of time in their models will be one tenth of a second (also known as a decisecond). This modelling decision is coherent with the original case study specification [5], where timing constraint values are mostly given in integer seconds, but timing measurement values (based on real world estimations) are given as floating point decimals (to 1 decimal place). Both types of timing values are modelled as `NAT`s in the Event-B models. All the subsequent documentation that accompanies their model refers to units of time (u.t), which are deciseconds, when timing values (variables and constants) are modelled using the in-built `NAT` set.

Even in this simple case, there is a problem of model validation and understandability. The modellers must continually remember to multiply by ten all timing values given in the requirements specification. Readers of the document who are trying to validate the model against their requirements must continually remember to divide by ten all timing values given in the Event-B model. This type of mental gymnastics is illustrated by the requirement:

If one of the three gears is not seen locked in the down position more than 10 seconds after stimulating the outgoing electro-valve, then the boolean output normal mode is set to false.

and the corresponding Event-B condition:

$$(currentTime > TimeStimulationExtendRetractEv + 100 \wedge extend\_EV = TRUE) \Rightarrow (anomaly = TRUE \vee gear\_extended\_ind = PositionsDG \times \{\texttt{TRUE}\})$$

A secondary concern with this modelling approach is maintainability. If we imagine that a new requirement is added that uses a time measured to 2 decimal places, it will now be necessary to globally change all time values to units of a hundredth of a second (centiseconds). This may not be as straightforward as just multiplying all integer `NAT` values by 10, as `NAT`s may also have been used to model other variable types in the system (and not just times). So we must be able to locate all time variable and constant definitions. This may be done by introducing naming conventions (such as starting every time with the letter 'T'); but such approaches are notoriously brittle to changes to systems and models as they evolve. Further, such naming conventions are not necessarily standard, and so composition and re-use of models which use them becomes non-trivial.

Using our approach, we re-use the deciseconds measurement model in the Event-B context shown in Figure 5.

```
CONTEXT
  C_Deciseconds
EXTENDS
  C_Measurements
CONSTANTS
  decisecond
  multiplyBy10
AXIOMS
  decisecond conversion function type   :   multiplyBy10∈ Values ⟶ Values
                                                                          //   Covert decisecond to Second -
                                                                          //   multiply by 10
                                          ∀ s, b·  s ∈ ℤ ∧ b ∈ ℤ ⇒
  decisecond conversion function def    :     multiplyBy10(MakeFloat( s↦ b))
                                                 = MakeFloat(s ↦ b+1)
                                                                            //   Multiply value by 10 -
                                                                            //   increment the base power
  decisecond type    :   decisecond ∈ Values ⟶ Measurements
                       ∀ v· v ∈ Values ⇒
  decisecond def    :      decisecond(v) = TimeDimension ↦ v  ↦  multiplyBy10
                                                                             //   decisecond measures time
END
```

Figure 5: The dimension model - `CONTEXT C_Deciseconds`

The expression, from the original model:

$$currentTime > TimeSimulationExtendRetractEv + 100$$

can then be replaced by the expression:

$$gt(currentTime \mapsto (AddMeasurements(TimeSimulationExtendRetractEv, Second(MakeFloat(1 \mapsto 1)))$$

The original model assumes that the variables *currentTime* and *TimeSimulationExtendRetractEv* are both times and that they have the same units. Further, it assumes that the value 100 is also a time given in the same units as the two variables. With our explicit approach, the fact that the two variables and one constant are all times will be checked by the Rodin tool as it verifies that the expression is well-defined. Further, there is no need to statically check that the units are homogeneous; if they are different then the conversions will be done implicitly. Finally, it is easier to validate our model against the original natural language requirements as the *10* units of time specified in *seconds* are modelled explicitly as

1×10^1 `Seconds`. (The *clumsy* syntax of our model will be improved in future work, where we model measurements and values using a theory plug-in for the Rodin tool.)

4.2 Overloading a *type/set/class* with values from a different scale of the same dimension

In '*Integrating Domain-Based Features into Event-B: A Nose Gear Velocity Case Study*' [21], we see a model which represents velocities using 3 different scales: Inch per millisecond (IPmS), Miles per hour (MPH), and Kilometres per hour (KPH). There is an explicit typing of different base measurements: `INCH`, `MILE`, `MILLISECOND` and `HOUR`. However, kilometres are not modelled explicitly as a type; but they are modelled implicitly in axioms *axm1* and *axm2*:

$$axm1 : kph \in \mathbb{N} \rightarrowtail KPH, axm2 : mphTokph \in MPH \twoheadrightarrow KPH$$

We also see an explcit type for each of the velocities `MPH` and `KPH`, but no explicit type for `IPmS`, which appears implicitly in axiom *axm6*.

$$axm6 : mph\_velo \in (INCH \times MILLISECONDS) \rightarrow MPH$$

Axiom *axm6* states only that the velocity value is calculated from the inches value and the milliseconds value, but there is no explicit statement that the length dimension is **divided by** the time dimension.

The explicit function (bijection) that maps `MPH` to `KPH` (`mphTokph`) implies that there is a dimensional equivalence (that of velocity), and we can therefore coherently compare `MPH` and `KPH` values. There is no explicit statement that `MILE`s and `INCH`es represent the same dimension (length); and that `MILLISECOND`s and `HOUR`s represent the same dimension (time). The explicit types - as defined in their model - will help us to formally express and validate the natural language requirements; but the lack of explicit dimensions can make the process complex. For example, the requirement:

> "*Estimated ground velocity of the aircraft is available only if it is within 3 KPH of the true velocity at some moment within the past 3s*"

contains the constant value 3 for both a velocity and for a time. The original specification of this behaviour[1] is given in Figure 4.2.

INVARIANTS

$$inv1 : esmt\_velo\_avlbl \in BOOL \land published\_Velocity \in KPH \land actual\_velocity \in KPH$$

$$inv2 : esmt\_velo\_avlbl = TRUE \Rightarrow mphTokph(esmt\_velocity) \in kph[kph^{-1}(actual\_velocity) - 3..kph^{-1}(actual\_velocity) + 3]$$

$$inv3 : esmt\_velo\_avlbl = TRUE \Rightarrow ms(modMaxBit(ms^{-1}(Millisec) - ms^{-1}(updateTime))) \in ms[0..3000]$$

$$inv4 : ms(modMaxBit(ms^{-1}(Millisec) - ms^{-1}(updateTime))) \notin ms[0..3000] \Rightarrow published\_Velocity = kph(0)$$

$$inv5 : actual\_velocity \neq kph(0) \land mphTokph(esmt\_velocity) \notin kph[kph^{-1}(actual\_velocity) - 3..kph^{-1}(actual\_velocity) + 3] \Rightarrow published\_Velocity = kph(0)$$

$$inv6 : actual\_velocity \neq kph(0) \land mphTokph(esmt\_velocity) = published\_Velocity \Rightarrow \left(\begin{array}{l} mphTokph(esmt\_velocity) \in kph[kph^{-1}(actual\_velocity) - 3..kph^{-1}(actual\_velocity) + 3] \land \\ ms(modMaxBit(ms^{-1}(Millisec) - ms^{-1}(updateTime))) \in ms[0..3000] \end{array} \right)$$

[1] We have highlighted the constant values that are linked to the natural language text.

The "3 KPH" from the requirement text is found in 6 locations - highlighted in yellow - in the formal model. The "3 s" from the requirement text is found in 3 locations - highlighted in blue - in the formal model. Having an implicit rather than explicit typing of the values "3" in the model compromises understandibility and introduces complexity. As such, the model is more difficult to maintain and extend.

Following this modelling method, the coherence of arithmetic operations over values cannot be checked automatically. For example, we have edited a sub-expression of *inv2* in order to mix velocity and time (written in red). The mismatch in types, in this case, will not be detected automatically because the Rodin toolkit considers the expression to be well-defined.

$$kph[kph^{-1}(actual\_velocity) - 3.. \quad \textbf{ms}^{-1} \ \textbf{(3000)} \quad + 3]$$

In their approach, all arithmetic is done over the `NAT` values in the domain of the typing functions. This facilitates the verification (proving) process, but hinders the validation process, and permits the modelling of mixed-type expressions that are "meaningless" in the real world.

This model also suffers from maintainability and scalability issues. Rather than explicitly modelling the scaling of values between only non-canonical and canonical forms, they provide an explicit mapping between 2 non-canonical forms, namely `MPH` and `KPH`. Such explicit mappings are good for models with few units of measure, but they create scaleability problems when multiple units of measure are defined. For example, with 3 unit lengths - miles, kilometres, inches - and 2 unit times - hours and seconds - we have 6 possible unit velocities. An explicit transformation function between each pair of unit velocities would require 30 different function definitions in total. It would be much better to require only a single explicit function for mapping a value in each new single dimension unit to the canonical value in the same unit dimension; and for the transformations of all other units to be done in an automated, implicit, fashion. In this way, the model is much more scaleable to the introduction of new units.

Finally, the model also introduces ambiguous, non-standard notation. The model's invariants contain the expression ms^{-1}. In standard canonical form this would represent a velocity (metres per second); however, due to an unfortunate choice of syntax, in their model it represents a mapping from a millisecond time to a `NAT` value. Such syntax clashes are inevitable when composing models, except if a standard formal ontology (as we propose) is used by all developers.

With our approach we would explicitly model the new `Velocity` dimension and the 3 different units using the `AXIOM`s as shown in Figure 6 :

```
velocity type              :  VelocityDimension ∈ Dimension
velocity length/time       :  VelocityDimension =
                              DivideDimensions (LengthDimension ↦ TimeDimension)
mph type                   :  mph ∈ Values ⟶ Measurements                    //  miles per hour
mph conversion function    :  mphTOmps ∈ Values ⟶ Values                     //  convert to metres per second
mph def                    :  ∀ v· v ∈ Values ⇒
                              mph(v) = VelocityDimension ↦ v  ↦  mphTOmps
kph type                   :  kph ∈ Values ⟶ Measurements                    //  kiometres per hour
kph conversion function    :  kphTOmps ∈ Values ⟶ Values                     //  convert to metres per second
kph def                    :  ∀ v· v ∈ Values ⇒
                              kph(v) = VelocityDimension ↦ v  ↦  kphTOmps
ipms type                  :  ipms ∈ Values ⟶ Measurements                   //  inches per millisecond
ipms conversion function   :  ipmsTOmps ∈ Values ⟶ Values                    //  convert to metres per second
ipms def                   :  ∀ v· v ∈ Values ⇒
                              ipms(v) = VelocityDimension ↦ v  ↦  ipmsTOmps
```

Figure 6: The dimension model - `CONTEXT C_Velocity AXIOMS`

Using the `Velocity` context, we can now re-write *inv6* from the original model as:

$$(neq(actual\_velocity \mapsto kph(0)) \land eq(esmt\_velocity \mapsto published\_velocity)) \Rightarrow$$
$$(gt(esmt\_velocity \mapsto SubtractMeasurement(actual\_velocity \mapsto kph(3)) \land$$
$$lt(esmt\_velocity \mapsto AddMeasurements(actual\_velocity \mapsto kph(3))$$

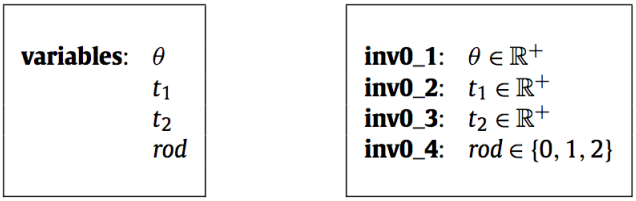
Coherency of the comparison functions (*neq*, *gt* and *lt*) and the arithmetic functions (*SubtractMeasurement* and *AddMeasurement*) is guaranteed by the welldefinedness of the expression, as checked by the Rodin tool.

4.3 Overloading a *type/set/class* with values from different dimensions

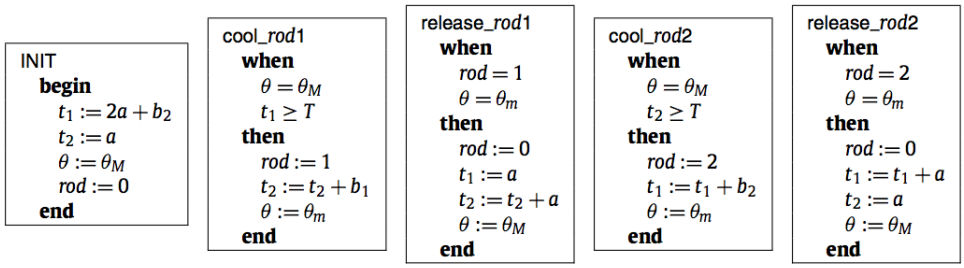
In '*Formalizing hybrid systems with Event-B and the Rodin Platform*' [26], we see the modellers overloading the positive reals to represent both temperature and time in a nuclear plant reactor cooling system:

> *We define some state variables: θ is the temperature of the reactor, t1 and t2 denote the time elapsed since rod1 or rod2 have been released.*

The modelling of the state of the real world measurements (temperature and time) and the discrete state of the rod is as follows:

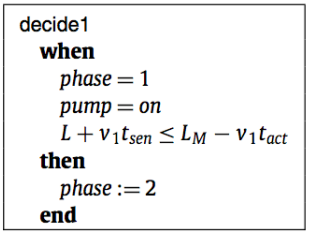


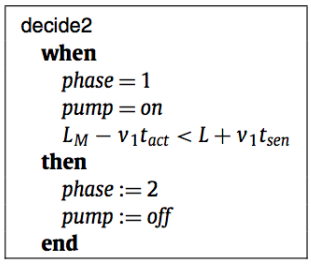
These variables are used in specifying the control events:



With regard to the risk of adding values with different dimensions, we see expressions, such as $2a + b_2$ and $t_2 + b_1$, where there is a risk that the operands are not coherent measures in the real world. The same risk is taken when comparing values (eg, $\theta = \theta_M$) and assigning values (eg, $\theta := \theta_m$). Reading the model, it is clear that the authors have taken great care in naming their variables in order to help avoid any incoherent combination of the real values. However, the additional burden on the modellers to check for and guarantee coherence becomes ever more significant as the model introduces implicit composition of dimensions. A good example is the pump controller model, where a variable v_1 has implicit dimension of length divided by time (a velocity):

In our approach, we already have explicit dimensions and measures for times and temperatures. The only additional extra dimension required by their model is that of velocity; and so we can re-use the model given in the previous section 4.2.





5 Application to other domains - currency example

The previous sections have demonstrated the utility of our approach with respect to modelling of the natural physical world. However, humans have also introduced abstract quantities into many domains, and these often suffer from the same modelling problems as for the concrete physical world. A typical example is one of currencies. In a *shopping/services on-line domain*, it is often possible to pay using different currencies (dollars, euros, etc...) in different subsystems. The role of the system would be to provide an ontology matching when combining such systems and domains [10]. We need to be able to check that we coherently perform numerical operations using currency values in the same way as for physical measurements. This would appear to be a trivial extension to our approach. However, it is made more complex by the conversion functions between currencies being variable (rather than constant, as for physical measurement conversions). In order to model this in Event-B, we are obliged to model the state of the system (in a machine).

In our example, customers (in a set C) and providers (in set P) are interacting, when customers are requesting services (in a set S). A service is provided by one and only one provider. A customer should first request a service; the provider sends him or her an invoice and finally the service is paid by the customer. When the service is paid, it is delivered.

We have three levels of machines[2] - M0, M00 and M000 - corresponding to a progressive modelling of the problem as a sequence of refinements.

Machine M0: The customer is requesting a service and the service is delivered by the provider. No expression of cost is given and no expression of failure in transaction is stated in our model. Two events model the possible actions operating on variables deliver and order; *Ordering_a_service* and *Serving_a_requested_service*. The invariant is simply stating that a service is corresponding to a request ($deliver \subseteq order$).

Machine M00 introduces the billing system in the process. A service should be paid before being delivered. New variables related to billing are introduced, as well as the actions of sending an invoice and paying an invoice. The invariant is simply stating the sequentiality of different actions and the strengthening of previous actions:

$$\begin{array}{l} inv1 : pay \in S \leftrightarrow C \\ inv2 : billing \in S \leftrightarrow C \\ inv3 : billing \subseteq order \\ inv4 : pay \subseteq billing \\ inv5 : deliver \subseteq pay \end{array}$$

[2] The full models for this case study - and for the previous measurement contexts - will be provided by the authors on request.

The machine has two new events which are modelling the two new variables: *Billing_a_requested_service* and *Paying_a_requested_service*. The event *Serving_a_requested_service* has a new guard which is strengthening the condition of service.

$inv1 : account \in U \rightarrow Values$	$inv10 : bills = dom(val)$
$inv2 : val \in B \nrightarrow Values$	$inv11 : bills = dom(cust)$
$inv3 : cust \in B \nrightarrow C$	$inv12 : bills = dom(prov)$
$inv4 : prov \in B \nrightarrow P$	$inv13 : bills = dom(ser)$
$inv5 : ser \in B \rightarrowtail S$	$inv14 : tarif \in S \rightarrow Values$
$inv6 : npay \in B \nrightarrow Values$	$inv15 : dom(billing) = ran(ser)$
$inv7 : date \in B \nrightarrow D$	$inv16 : dom(npay) \subseteq dom(ser)$
$inv8 : t \in B \nrightarrow (Values \rightarrow Values)$	$inv17 : dom(pay) = ser[dom(npay)]$
$inv9 : bills \subseteq B$	$inv19 : dom(npay) \subseteq bills$

Machine M000 refines machine M00 by *superposing* new variables in the current events. We introduce new variables such as the variable *bills* which models the set of already existing bills. The main critical property is the invariant *inv*22 which states that the value paid by the customer to the provider corresponds to the value in the currency of the customer: $cpay(cust(b) \mapsto ser(b)) = t(b)(val(b))$. $t(b)$ is the translation of the value of the bill in the currency of the provider ($val(b)$) and in the variables, we record the current translation used while paying.

$$
\begin{array}{l}
inv20 : \forall b \cdot b \in bills \wedge b \in dom(npay) \Rightarrow val(b) = npay(b) \\
inv21a : cpay \in C \times S \nrightarrow Values \wedge dom(npay) = dom(date) \\
inv21b : dom(t) = dom(date) \wedge dom(npay) = dom(t) \\
inv22 : \forall b \cdot \left(\begin{array}{l} b \in bills \wedge b \in dom(date) \\ \Rightarrow \\ cust(b) \mapsto ser(b) \in dom(cpay) \wedge cpay(cust(b) \mapsto ser(b)) = t(b)(val(b)) \end{array} \right) \\
inv23 : dom(t) = dom(date) \\
inv24 : dom(npay) = dom(t)
\end{array}
$$

In this simple case study we see that the static approach for `Measurements`, where the conversion functions are constant, has the potential to be extended to a more dynamic approach (as for currency payments in the example) where the conversion functions are variable. It is currently work-in-progress to extend our methodology to handle the dynamic case.

6 OO Design and Implementation

6.1 Inheritance Hierarchy

We chose to implement our `Measurements` model in Java because of the rich typing and polymorphic semantics, together with type safety for a reasonable subset of the language [9]. The diagram, in Figure 7, is a partial representation of the inheritance hierarchy in our object oriented implementation (using Java). There are seven single dimension measurements: Mass, Length, Time, Temperatures, Luminosity,

Current and Substance, each of which has a standard unit class: Kilogram, Metre, Second, Kelvin, Candela, Ampere and Mole. These are in the base library of Measurements that provide the foundations for constructing three kinds of new classes:

- **Non standard single dimensional measurements**, like Centimetre, Kilometre, Mile, Hour and Decisecond (in the diagram).
- **Standard multiple dimensional measurements**, like Velocity in MetresPerSecond (in the diagram).
- **Non standard multiple dimensional measurements**, like Velocity in MilesPerHour (in the diagram).

Developers are free to add new Measurements (of all 3 types) by extending the class hierarchy. They must execute the Measurement unit tests (see section 6.3) on instances of their new classes in order to (partially) verify that the new classes respect the generic requirements as specified in the Event-B context models (such as associativity and commutativity of the addition operation/method).

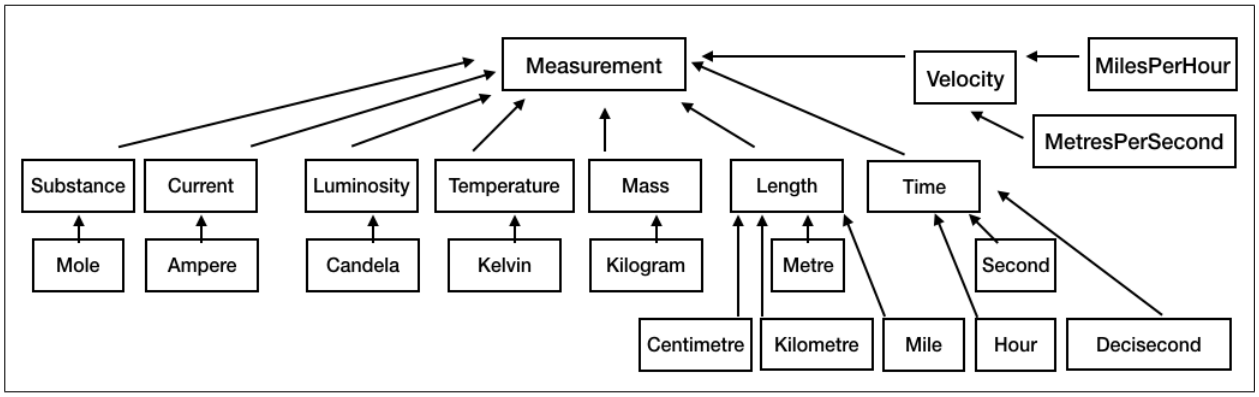


Figure 7: (Incomplete) Measurement Class Inheritance Hierarchy

6.2 The Java Code

The abstract behavior is first modelled in the class `Measurement.java`, below. Concrete measurement classes (in the standard seven dimensions and all possible combinations) are required to override the methods of this general class:

- the standard four numerical operations, with exceptions thrown for addition and subtraction of measurements in different dimensions;
- a method to permit the simple scaling of a measurement;
- two methods for converting to/from the standard unit of measurement for the given dimension;
- two methods (static and non-static) to check if measures have the same dimension
- two constructors (including a random constructor)
- standard getter methods, toString, equals and hashcode

The implementation of each base dimension measurement is now straightforward. For example, the Java code for the `Second` class is given, below.

```
public  class Measurement {

public static final int NUMBER_OF_DIMENSIONS =7;
protected int [] dimensions;
protected  double VALUE;

public Measurement (int [] dimensions, double value) throws IllegalArgumentException{ …
public Measurement (Random rng) { …
public double getValue() {…
public int[] getDimensions() {…
public String getUnit() {…
public String getAbbreviation() {…
public Measurement toStandard() {…
public static Measurement convertMeasurement(Measurement measurement) throws IllegalArgumentException{…
public boolean sameDimensionsAsObject (Measurement measurement){…
public static boolean  sameDimensionAsClass (Measurement measurement) {…
public void scale(double scalar) {…
public void add(Measurement measurement) throws IllegalArgumentException {…
public void subtract(Measurement measurement) throws IllegalArgumentException {…
public Measurement multiply(Measurement measurement) {…
public Measurement divide(Measurement measurement){…
…
}
```

```
import implementations.standardMeasurements.StandardMeasurement;
import enumerations.BaseUnits;

public class Second extends StandardMeasurement{

    public Second (){
        super (new int [] {0,0,0,0,0,0,1}, 0);
    }

    public Second (double value){
        super (new int [] {0,0,0,0,0,0,1}, value);
    }

    public Second (double value, double delta){
        super (new int [] {0,0,0,0,0,0,1}, value, delta);
    }

    public String get_unit() {
        return BaseUnits.TIME.unit;
    }

    public String get_abbreviation() {
        return BaseUnits.TIME.abbreviation;
    }

    public String toString (){
        return get_value()+get_abbreviation();
    }
}
```

The implementation of more complex single and multiple dimensional measurements is slightly more complicated as it requires care when writing the methods for converting to/from the standard units. We note that this is a simple scaling function (except in the case of temperatures). The required properties (as specified in the Event-B theorems) are then used to drive the coding and execution of unit tests on the new measurement classes.

6.3 Java - unit tests from Event-B theorems

Following the approach in [11], we use the formal models to drive the generation of test cases, rather than refining them towards an implementation. Thus, we identify theorems in the contexts (that have been proven by the Rodin tool) and demonstrate that the properties that they specify are also valid for the Java implementation (through testing). For example, the commutativity and transitivity of the addition of measurements (of the same dimension) can be verified automatically by the RODIN tool, but needs to be tested in the Java implementation.

The technique that we employ is very simple, and based on random test generation combined with the JUnit tool. The abstract tests are coded in the JUnit_Measurement class, which must be extended by a unit test class for each new Measurement class. For example, the Second class has a test class JUnit_-Second which extends JUnit_Measurement. The tests use random constructors to check that the required properties are respected for a *large number of randomly generated* test cases[3] This does not guarantee that the properties are respected for *all* the new measurement values, but they reassure the developers that there is a very high probability that their code is correct (with respect to the properties being tested).

7 Ongoing and Future Work

We are currently pursuing three areas of research which extend this work:

I. Accuracy of measurement - In this paper we have not addressed the need to model the fact that values and measurements in our systems are actually approximations to reals. In certain types of (relatively simple) systems this can lead to unpredictability and chaotic behaviour [12]. Consequently, in order to reason about the behaviour of such systems, it is necessary for values to modelled within a certain error (within a bounded interval). Fortunately, there is already much existing work on interval values and interval arithmetic [22, 18, 18, 15, 7]. We have already extended our Java measurement library to include interval value arithmetic. We hope to report on this in a future paper.

II. Theory of measurements - Rodin provides a *Theory plug-in* that facilitates definitions of mathematical and prover extensions [16]. It should be relatively straightforward to introduce a theory of `Measurements` using the current algebraic specifications in our `Values`, `Dimensions` and `Measurements` contexts. However, it seems wise to wait until our work on modelling values as intervals is complete and validated before we attempt to build a theory extension for interval measurements. It should be noted that such a theory extension would help to tidy up our current *clumsy* notation, and would facilitate more automated proof of simple mathematical properties of measurements.

III. Re-engineering case studies for validation and extension of our approach - It is important to validate our approach using real world case studies. We are currently re-engineering a number of case studies that have used measurements implicitly to adopt our explicit measurement modelling approach. We believe that these re-engineered models - thanks to the explicit modelling of measurements - will be easier to validate, more amenable to automated coherency checking, and easier to maintain. One of the case studies that we are currently re-engineering is the currency example - mentioned previously in this paper - where there is a need to model and reason about dynamic rather than static unit conversion.

[3] We can run millions of tests in a matter of seconds.

8 Conclusions

We have motivated the need for explicit modelling of dimensions of physical measurements during formal development of systems. We have demonstrated one approach for such modelling using Event-B contexts, that meets all our formal engineering requirements. We have compared this approach with other case studies in which the modelling of dimensions is implicit. Java implementations of the Event-B specifications demonstrate the feasibility of the approach. Java unit tests are then derived from the Event-B specifications.

Acknowledgement

The authors acknowledge the support provided by the French ANR project IMPEX (13-INSE-0001)

References

[1] Jean-Raymond Abrial (2010): *Modeling in Event-B - System and Software Engineering*. Cambridge University Press. Available at `http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=9780521895569`, doi:10.1017/CBO9781139195881

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta & Laurent Voisin (2010): *Rodin: an open toolset for modelling and reasoning in Event-B*. Int. J. Softw. Tools Technol. Transf. 12, pp. 447–466, doi:10.1007/s10009-010-0145-y.

[3] Jean-Raymond Abrial & Stefan Hallerstede (2007): *Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B*. Fundam. Inf. 77(1-2), pp. 1–28.

[4] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen & Guy L Steele Jr (2004): *Object-oriented units of measurement*. ACM SIGPLAN Notices 39(10), pp. 384–403, doi:10.1145/1028976.1029008.

[5] Frédéric Boniol & Virginie Wiels (2014): *The Landing Gear System Case Study*. In: *ABZ 2014: The Landing Gear Case Study*, Springer, pp. 1–18, doi:10.1007/978-3-319-07512-9_1.

[6] Jonathan Bowen & Victoria Stavridou (1993): *Safety-critical systems, formal methods and standards*. Software Engineering Journal 8(4), pp. 189–209, doi:10.1049/sej.1993.0025.

[7] Hervé Brönnimann, Guillaume Melquiond & Sylvain Pion (2003): *The Boost interval arithmetic library*. In: *Real Numbers and Computers*, pp. 65–80.

[8] Jonathan Cooper & Steve McKeever (2008): *A model-driven approach to automatic conversion of physical units*. Software: Practice and Experience 38(4), pp. 337–359, doi:10.1002/spe.828.

[9] Sophia Drossopoulou, Susan Eisenbach & Sarfraz Khurshid (1999): *Is the Java type system sound?* TAPOS 5(1), pp. 3–24.

[10] Jérôme Euzenat, Pavel Shvaiko et al. (2007): *Ontology matching*. 18, Springer, doi:10.1007/978-3-642-38721-0.

[11] J. Paul Gibson, Eric Lallet & Jean-Luc Raffy (2011): *Formal Object Oriented Development of a Voting System Test Oracle*. Innovations in Systems and Software Engineering (Special issue UML-FM11) 7(4), pp. 237–245, doi:10.1007/s11334-011-0167-y.

[12] James Gleick (2011): *Chaos: Making a new science (Enhanced edition)*. Open Road Media.

[13] David Goldberg (1991): *What every computer scientist should know about floating-point arithmetic*. ACM Computing Surveys (CSUR) 23(1), pp. 5–48, doi:10.1145/103162.103163.

[14] Thomas R Gruber & Gregory R Olsen (1994): *An Ontology for Engineering Mathematics*. KR 94, pp. 258–269.

[15] Timothy Hickey, Qun Ju & Maarten H Van Emden (2001): *Interval arithmetic: From principles to implementation*. Journal of the ACM (JACM) 48(5), pp. 1038–1068, doi:10.1145/502102.502106.

[16] Thai Son Hoang, Laurent Voisin, A. Salehi, Michael J. Butler, Toby Wilkinson & N. Beauger (2017): *Theory Plug-in for Rodin 3.x*. CoRR abs/1701.08625. Available at `http://arxiv.org/abs/1701.08625`.

[17] Lingxiao Jiang & Zhendong Su (2006): *Osprey: a practical type system for validating dimensional unit correctness of C programs*. In: Proceedings of the 28th international conference on Software engineering, ACM, pp. 262–271, doi:10.1145/1134285.1134323.

[18] R Baker Kearfott (1996): *Interval computations: Introduction, uses, and resources*. Euromath Bulletin 2(1), pp. 95–112.

[19] Andrew Kennedy (1996): *Programming languages and dimensions*. Ph.D. thesis, University of Cambridge, Computer Laboratory.

[20] Amel Mammar & Régine Laleau (2014): *Modeling a landing gear system in Event-B*. In: International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, Springer, pp. 80–94, doi:10.1007/978-3-319-07512-9_6.

[21] Dominique Méry, Rushikesh Sawant & Anton Tarasyuk (2015): *Integrating Domain-Based Features into Event-B: A Nose Gear Velocity Case Study*. In Ladjel Bellatreche & Yannis Manolopoulos, editors: Model and Data Engineering - 5th International Conference, MEDI 2015, Rhodes, Greece, September 26-28, 2015, Proceedings, Lecture Notes in Computer Science 9344, Springer, pp. 89–102, doi:10.1007/978-3-319-23781-7_8.

[22] Ramon E. Moore (1962): *Interval Arithmetic and Automatic Error Analysis in Digital Computing*. Ph.D. thesis, Department of Computer Science, Stanford University.

[23] TJ Quinn (1995): *Base units of the Système International d'Unités, their accuracy, dissemination and international traceability*. Metrologia 31(6), p. 515, doi:10.1088/0026-1394/31/6/011.

[24] V Frederick Rickey (1992): *How Columbus encountered America*. Mathematics Magazine 65(4), pp. 219–225, doi:10.2307/2691445.

[25] Brian J Sauser, Richard R Reilly & Aaron J Shenhar (2009): *Why projects fail? How contingency theory can provide new insights - A comparative analysis of NASAs Mars Climate Orbiter loss*. International Journal of Project Management 27(7), pp. 665–679, doi:10.1016/j.ijproman.2009.01.004.

[26] Wen Su, Jean-Raymond Abrial & Huibiao Zhu (2014): *Formalizing hybrid systems with Event-B and the Rodin platform*. Science of Computer Programming 94, pp. 164–202, doi:10.1016/j.scico.2014.04.015.

[27] Ambler Thompson & Barry N Taylor (2008): *Guide for the Use of the International System of Units (SI)*. Special Publication (NIST SP)-811, doi:10.6028/NIST.SP.811e2008.

[28] Hernán Wilkinson, Máximo Prieto & Luciano Romeo (2005): *Arithmetic with Measurements on Dynamically-typed Object-oriented Languages*. In: Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05, ACM, New York, NY, USA, pp. 292–300, doi:10.1145/1094855.1094964.